A Component Model Engineered with Components and Aspects

Lionel Seinturier, Nicolas Pessemier, Laurence Duchien, Thierry Coupaye

A Component Model Engineered with Components and Aspects

Lionel Seinturier[1], Nicolas Pessemier[1],
Laurence Duchien[1], and Thierry Coupaye[2]

[1] INRIA Futurs - LIFL, Projet Jacquard/GOAL
Bâtiment M3, 59655 Villeneuve d'Ascq, France
{seinturi, pessemie, duchien}@lifl.fr
[2] France Telecom R&D
28 chemin du Vieux Chêne, BP98
38243 Meylan, France
Thierry.Coupaye@francetelecom.fr

Abstract. This paper presents AOKell, a framework for engineering component-based systems. This framework implements the Fractal model, a hierarchical and dynamic component model. The novelty of this paper lies in the presentation of AOKell, an implementation of the Fractal model with aspects. Two dimensions can be isolated with Fractal: the functional dimension, which is concerned with the definition of application components, and the control dimension, which is concerned with the technical services (e.g. lifecycle, binding, persistence, etc.) that manage components. The originality of AOKell is, first, to provide an aspect-oriented approach to integrate these two dimensions, and second, to apply a component-based approach for engineering the control dimension. Hence, AOKell is a reflective component framework where application components are managed by other, so-called, control components and where aspects glue together application components and control components.

1 Introduction

Software components are more and more used in various application domains. This trend is supported by the fact that many component models are available, coming either from the industry such as Sun EJB [1], Microsoft .NET/COM+, OMG CCM [2], OSGi [3], or from research teams (e.g. ArchJava [4], Fractal [5], FuseJ [6], K-Component [7], OpenCOM [8]).

In our opinion, the domain of component-based software engineering is characterized by two main requirements: the need for components goes beyond the boundaries of programming languages, and components need to be used in various execution contexts, such as embedded applications with strong constraints in terms of memory footprint and execution costs, information systems hosted on application servers, or grid computing. In this paper, we argue that the challenge for component models is to be able to handle these requirements. So far, existing component frameworks are mostly seen as closed, black box entities

I. Gorton et al. (Eds.): CBSE 2006, LNCS 4063, pp. 139–153, 2006.

which provide artefacts to design and program applications with components. The components are handled by the framework, which provides a set of services to manage these application components. Yet, this set of services is most of the time closed. This is the case for example, with the EJB [1] component model, where new services cannot be added to the container.

In this paper we propose AOKell, which is an open implementation in Java of the Fractal component model. By implementation, we mean a software infrastructure for defining and executing components. The implementation is open in the sense that the services provided by the AOKell framework are fully accessible and programmable. By giving programmers a way to engineer these services, AOKell eases the task of adapting component-based applications to different execution contexts. This approach also fosters the development of various forms of control for components such as the ones needed to program self healing components, self-testing components, or components that carry their proofs or their specifications. Two main software techniques are used to engineer these services: components and aspects. Both the applications and the services provided to the applications are designed and implemented with components. Aspects glue together these two dimensions. This paper presents the design and the implementation of AOKell in Java with the AspectJ [9] aspect-oriented language. Although we do not report on it in this paper, AOKell has also been ported to the .NET platform [10].

The paper is organized as follows. Section 2.1 presents the background of this work: the Fractal component model and aspect-oriented programming. Section 3 is the core of the paper and presents the design of the AOKell framework. We show how aspects are used in AOKell (section 3.1) and we present the model for customizing the control dimension (section 3.2). Section 4 reports some performance measurements. Section 5 compares AOKell to similar existing projects. Section 6 concludes this paper and presents our future work directions.

2 Background

2.1 The Fractal Component Model

The Fractal component model [5] is a general model for developing component-based systems. The model is sufficiently open to accommodate the needs of various application domains. For example, the model has been used to implement applications for grid computing [11], operating systems [12], the GoTM transaction monitor [13], a version of the JORAM [14] JMS [15] server and the Speedo [16] JDO [17] persistence framework.

AOKell, the framework presented in this paper, is an implementation of the Fractal component model for the Java programming language. Implementations exist in other programming languages: FracTalk in Smalltalk, Plasma in C++, Think [12] in C, FractNet [10] for the .NET platform. Two additional implementations in Java exist: Julia, which is the reference implementation, and ProActive, which is an implementation for grid computing. Information about these

implementations can be found on the Fractal web site[1]. As this will be explained in section 3, the added value of AOKell compared to these implementations is to be based on some concepts of aspect-oriented programming and to introduce the notion of a control component.

Fractal is a hierarchical and dynamic component model. The model is hierarchical in the sense that a component can be composite or primitive. A composite component contains other primitive or composite components. A primitive component is the smallest unit of code packaged as a component. The model is dynamic in the sense that the software architecture of a Fractal application can be manipulated at runtime: components can be created, containment hierarchies can be modified, and bindings (which are communication paths between components) can be set and unset. Components can be shared which means that a component can be included in several non nested composite components. This feature allows designing as components shared resources such as pools (for threads, network sockets, etc.).

Two dimensions can be isolated in the Fractal component model: the functional dimension and the control dimension.

Functional Dimension. The functional dimension is concerned with programming the core functionalities of the application. Besides the notion of a component, which can be primitive or composite, two main artefacts are provided to engineer the functional dimension: interface and binding.

An interface is an access point to a component and supports a finite set of operations. An interface can be of two kinds: server and client. Server interfaces correspond to the services provided by the components, whereas client interfaces correspond to the ones required by the components.

A binding is a communication path between two components, more precisely between a client interface and a server interface. Bindings can be dynamically set and unset to adapt, at runtime, the architecture of the application. The default semantics for the communication in a binding is that of a local method call. However, Fractal components can accommodate various other communication modes such as remote method call, asynchronous message passing, publish/subscribe.

Several other artefacts are provided such as the notion of a template. A template is an existing component assembly that can be cloned. Templates are a powerful means of instantiating, in just one step, complex software architectures containing several components and bindings.

The Fractal component model is associated with an API. The implementations of the model may conform to one of the levels defined in the Fractal Specifications [18], i.e. implementing the whole API is not mandatory. One of the tools worth noticing is Fractal ADL which is an architecture description language (ADL). Assemblies of components can be defined with this XML-based language, which is a front-end for the Fractal API. All architecture descriptions written with Fractal ADL are translated, either statically or dynamically, into series of calls to the API. These calls install the assemblies described with Fractal ADL.

[1] `http://fractal.objectweb.org`

The next piece of XML code illustrates the syntax of Fractal ADL. This sample defines one composite component (`HelloWorld`) and two primitive ones: `client` (line 3) and `server` (line 8). `HelloWorld` provides the `run` interface (line 2). This interface is bound (line 12) to the `run` interface provided by `client`. The `server` component provides a `s` interface (line 9), which is bound (line 13) to the `s` interface requested (line 5) by `client`.

```
 1 <definition name="HelloWorld">
 2   <interface name="run" signature="Runnable" role="server"/>
 3   <component name="client">
 4     <interface name="run" signature="Runnable" role="server"/>
 5     <interface name="s" signature="IService" role="client"/>
 6     <content desc="ClientImpl"/>
 7   </component>
 8   <component name="server">
 9     <interface name="s" signature="IService" role="server"/>
10     <content desc="ServerImpl"/>
11   </component>
12   <binding client="this.run" server="client.run"/>
13   <binding client="client.s" server="server.s"/>
14 </definition>
```

Control Dimension. The control dimension of the Fractal component model is concerned with the supervision and the management of functional components. This dimension provides the services to handle components. The range of services incorporated into the control dimension can vary from basic services such as managing component names, to lifecycle services, or to more complex services such as persistence or transaction services. The control dimension plays a role rather similar to the one played by containers in component models such as EJB [1], except that this control dimension is open and fully programmable with Fractal. Two main artefacts are provided to engineer the control dimension: *membrane* and *controller*.

Each functional component is associated with a membrane. A membrane is composed of a set of smaller units, called controllers. A controller implements a particular control function and is associated to an interface. Controllers may either provide new functionalities to components, such as the ability to set or unset binding, or control existing functionalities, such as intercepting requests or blocking calls on a stopped component.

The Fractal Specifications [18] defines seven control interfaces. However this set is not closed and programmers can still develop their own control interfaces. Furthermore, although the signatures of these interfaces are defined in the specifications, their semantics is only weakly specified. The idea is to accommodate various implementations tailored to developers needs.

Among the seven predefined Fractal control interfaces, three are defined for managing component attributes, component bindings (with methods for setting, unsetting, retrieving and listing bindings), and component lifecycles (starting and stopping a component). Two additional control interfaces are available

for managing containment hierarchies: the content control interface manages (adding, removing, listing) sub-components contained in a composite, and the super control interface manages the super components attached to a component. The factory control interface is available for cloning a template. Finally, the component control interface is available for retrieving the basic information about a component such as the list of interfaces. This interface is similar to the `IUnknown` interface of the COM component model.

2.2 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) [19] is a software engineering technique for modularizing applications with many concerns. The general idea of AOP is that, whatever the domain, applications tend to be decomposed according to a dominant concern. The concerns which do not fit into this decomposition cannot be cleanly modularized.

This issue is illustrated with the well-known example of the Tomcat servlet server where some concerns such as XML parsing are cleanly modularized, whereas others, such as user session management, are implemented in many different classes. This leads to code that is said to be scattered (the implementation of a concern is scattered around several different locations), and tangled (a same piece of code mixes different concerns). AOP aims at providing solutions for untangling and unscattering applications. The notion of an aspect is available to modularize such concerns, which are said to be crosscutting. Several languages and frameworks such as AspectJ [9], JBoss AOP [20], AspectWerkz [21], JAC [22] or JAsCo [23] are available for programming aspect-oriented applications.

The AspectJ language has been chosen to develop the aspects needed by AOKell. This choice has been motivated by the fact that AspectJ is a stable and mature project, well integrated with widely used IDEs such as Eclipse. Also the fact that AspectJ currently provides features for compile-time and load-time weaving, allows covering a wide range of needs.

3 The AOKell Framework

AOKell is our implementation of the Fractal component model for the Java language. The functional dimension of a component-based application with AOKell strictly conforms to the Fractal model. By this way, AOKell can execute any Fractal system. AOKell differs from other existing implementations of the Fractal model by relying on aspects and components for engineering the control dimension, i.e. the services provided to functional components. By providing these two advanced software engineering techniques, we hope to promote flexibility and to allow adapting component-based applications to execution contexts with various and changing constraints.

Section 3.1 describe the structure of a component with AOKell and explain the role devoted to aspects. Next, section 3.2 presents the concepts which have been set up for "componentizing" the membranes.

3.1 Integrating the Control Dimension with Aspects

This section describes how aspects are used in AOKell to integrate control services into components. Section 3.2 will elaborate on the way these control functions are designed and implemented.

Component models such as EJB or CCM provide an architecture where components are hosted by containers that provide technical services. For example, the EJB specifications [24] define services for managing security, transaction, persistence and lifecycle. Most of the time, this set of services is closed and hard-coded in the container. One exception is the JBoss J2EE application server [25] where services can be wrapped and accessed with aspects defined with the JBoss AOP framework [20].

The general idea illustrated with the case of the JBoss server is that aspects, while providing a way for modularizing crosscutting concerns, allow smoothly integrating a concern into applications. This leads to a common practise of AOP: the aspect modularizes a given concern, and either implements it directly, or delegates it to an external module. The separation of concern is almost optimal in the sense that the aspect is only concerned with the logic for integrating the concern into the application.

This pattern is used with AOKell to integrate the control logic into components. More precisely, each control function (a so-called controller in Fractal terms) is associated with an aspect which is responsible for integrating this logic into components. This solution is illustrated in figure 1.

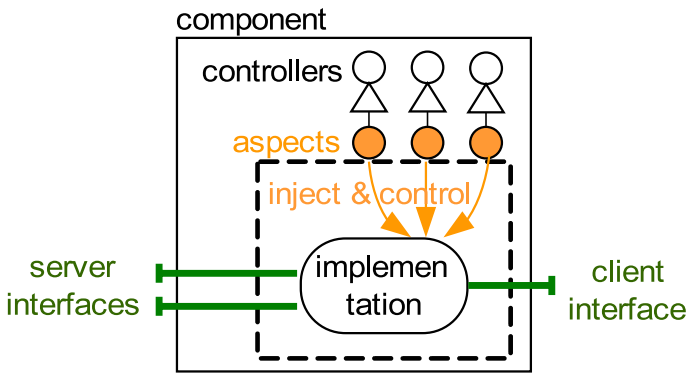

Fig. 1. Structure of a component with AOKell

The integration performed by aspects relies on two mechanisms: feature injection and behavior extension. The first mechanism is known, in AspectJ, under the term inter-type declaration (ITD). With ITD, aspects can declare members (methods and/or fields) to be injected into the classes, in our case into component implementations. All existing control interfaces are injected with this ITD mechanism.

The second mechanism is known, in AspectJ, under the term code advising. Aspects define so-called pointcuts and advice code. Pointcuts pick out a set of join points, which corresponds to the points of the program execution where the aspect needs to be executed. The advice code is a piece of code which will be executed at these points. Code advising is used in AOKell to intercept operation calls and executions. For example, when controlling a component, the lifecycle controller may reject operation executions while the component has not been started. This feature is implemented by defining, in the aspect associated to the lifecycle controller, pointcuts and pieces of advice code.

3.2 Componentized Membranes

The previous section showed how aspects are used with AOKell to integrate the control logic into components. This section elaborates on the way this control logic is designed and implemented.

We have seen that the control logic is defined in the Fractal component model with a membrane composed of controllers, each one being specialized with a particular control mechanism (binding management, lifecycle, etc.). Far from being autonomous, these controllers need to collaborate to achieve the global control function assigned to the membrane. For example, when starting a composite component, the content of this composite needs to be traversed to recursively start sub-components[2]. This implies that the lifecycle controller depends on the content controller. Several other similar dependencies exist between controllers. For clarity sake, we omit details for all these dependencies, which come from the semantics assigned to controllers. Readers can find them in [26].

However, the fact that these dependencies between controllers are hidden and not clearly expressed prevent developers from reusing controllers independantly. Our idea is to apply to the design of the control layer the same principles which were applied to the application layer: engineer the control with components. By "contractually specifying the interfaces" [27] of these control components, we hope to foster their reuse, to clarify the architecture of the membrane, and to ease the development of new ones. By supplying a component-based approach for engineering the control layer, we also hope to obtain gains in terms of flexibility: it will be easier to develop new control layers and thus to adapt applications to execution contexts with different characteristics in term of resource management (memory, threads, etc.).

As a consequence, AOKell is a framework where the concepts of a component, of an interface and of a binding are used to engineer the functional dimension and the control dimension as well. A control membrane with AOKell is a composite component providing the control interfaces associated to that membrane. This composite contains sub-components. Each sub-component implements the control functionality associated to a controller. As explained in the previous section, this component is associated to an aspect that integrates this control logic into application level components. Furthermore, these sub-components are

[2] Note that this is not a formal obligation. One may design a control function where the starting is not recursive.

bound together according to their dependencies. Figure 2 summarizes these elements. For clarity sake, the control membrane for the third component has been omitted.

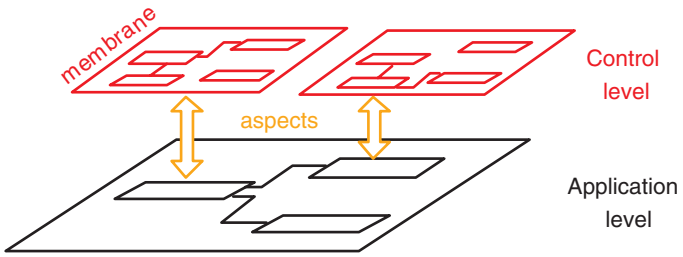

Fig. 2. AOKell component layers

The most widely used control membrane in Fractal applications is the one associated with primitive components. The architecture of this membrane is illustrated in figure 3. This membrane provides five controllers, for managing the lifecycle (`LC`), the bindings (`BC`), the component name (`NC`), the super components (`SC`) and a controller (`Comp`) implementing the general `Component` interface, which is available for all Fractal components. As a matter of convention, provided interfaces are drawn on the left side of the components, and required interfaces are on their right side. Bindings represent communication paths between the controllers.

The architecture presented in figure 3 illustrates that the control function for primitive components is not simply realized by five isolated controllers, but is the result of the collaboration of these five controllers. Compared to a purely object-oriented approach, a component-based solution for the implementation of control

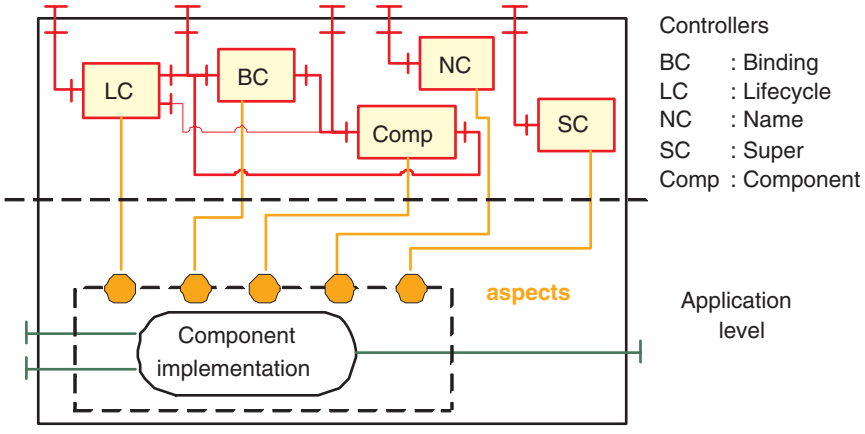

Fig. 3. Primitive membrane: control level for primitive components

membranes allows describing explicitly the dependencies between controllers. New control membranes can be developed by extending existing ones, or simply by developing a whole new architecture.

The benefits of engineering the control dimension with components have been experimented by implementing the Dream framework [28]. Dream is a framework for developing middleware platforms with Fractal. The purpose is to ease the development of middleware by providing a library of component with advanced control functionalities. For example, Dream provides a membrane to define active components, i.e. components with threads or pools of threads to handle their requests. Based on these membranes, a version of the JORAM [14] JMS [15] server has been developed with Dream. Basically, implementing the Dream framework with AOKell consists in implementing a component-based version of the controllers and of defining the architecture of the membranes.

4 Performance Evaluation

This section evaluates the cost of running a component based application with AOKell. We are mainly interested in measuring the cost induced by the component framework and the componentization of membranes. To do so, we compare an application developed with AOKell with the same one developed with a pure object-oriented approach.

AOKell is written in Java and uses AspectJ 1.2.1. The AOKell source code size is 12,604 lines with 104 classes and 13 aspects[3]. Other technical details can be found in [26]. AOKell has also been ported to the .NET platform [10]. For this porting, AspectJ has been replaced by AspectDNG [29].

The tests are conducted with a simple application containing two components: a client component and a server component. The server component provides an interface with eight methods. Each method owns a different signature, either without parameters, or with primitive parameters, or with object references parameters, and/or with return types.

The measures are done on a 2Ghz Pentium 4 PC running Windows XP Pro and Sun JDK 1.5.0. A warm-up phase is performed before taking measures to avoid bootstrapping and class loading costs induced by the JVM. The test consists of series of calls emitted from the client component to the server component. In table 1, the figures correspond to the times taken by 8,000,000 calls (1,000,000 per method defined in the interface provided by the server component). The given figures correspond to the average value of 4 runs.

Table 1 presents the result obtained for this microbenchmark with five different techniques.

- Fractal/Julia: this is a component-based Fractal implementation of the microbenchmark. This version is linked with the Julia (version 2.1.1) reference implementation of the Fractal Specifications.

[3] AOKell can be downloaded from `http://fractal.objectweb.org`

Table 1. Cost of invoking and executing an operation (x 8,000,000)

	Operation execution time	
	without interception	with interception
Pure Java 1.5.0	178ms	
AspectJ 1.2.1		209ms
Fractal/Julia 2.1.1	237ms	515ms
Fractal/AOKell 1.1	215ms	559ms
JBoss AOP 1.1.1		1046ms

- Fractal/AOKell: this Fractal version of the microbenchmark is linked with the AOKell implementation presented in this paper. These two last versions allow comparing a purely object-oriented implementation of the control dimension (Fractal/Julia) with an implementation where the control dimension is componentized (Fractal/AOKell).
- Java: this is a pure object-oriented Java implementation. No components are involved. The client and the server are Java objects. This implementation gives a reference to evaluate the cost of running a componentized application.
- AspectJ: this version is implemented with AspectJ version 1.2.1. No components are involved. The client and the server are Java objects. The server object is advised by an empty around advice. This version gives a clue on the cost of intercepting a method with AspectJ.
- JBoss AOP: this version is implemented with the JBoss AOP [20] (version 1.1.1) framework for dynamic AOP. No components are involved. The client and the server are Java objects. The server object is advised by an empty around advice. This version gives a clue on the cost of intercepting a method with JBoss AOP.

We saw in section 3.1 that controllers may, via aspects, either inject new features or modify the behavior of components by intercepting existing features. The microbenchmark reported in table 1 provides a measure of the interception cost of both Fractal versions.

Control without interception. When compared to the Java implementation, the AOKell version is 21% costlier. The main reason is that the binding between the client and the server component is dynamic: before each call, the reference to the target server component must be resolved. This ensures that at any time the architecture is modified, the communication path between components will be updated accordingly. We believe that this penalty is acceptable compared to the benefits of having a component architecture dynamically updatable.

The figures given in table 1 show that AOKell performs better than Julia. We believe that this is due to the way controllers are implemented in Julia: a mixin mechanism is provided to modularize the different concerns addressed by each controller. When mixed together, these different pieces of code are assembled in a class which contains more indirections than the AOKell version where controllers have been implemented directly. Compared to Julia controllers, AOKell

controllers are then less modular in terms of separation of concerns, but they are implemented as components (Julia controllers are objects) and they perform slightly better.

Control with interception. The interception costs reported in the second column of table 1 is due to the Fractal lifecycle controller. The purpose of this controller is to ensure that a call cannot be issued on a stopped component. This mechanism is implemented in Julia by engineering the bytecode of components with the ASM library [30], and in AOKell with AspectJ. When the interception mechanism is activated, the figures in table 1 shows that, compared to Julia, the overhead of running AOKell is 8.5%. This is mainly due to the use of AspectJ compared to that of ASM. In our opinion, this penalty is acceptable compared to the benefits of a high level language such as AspectJ compared to a bytecode engineering library such as ASM.

5 Related Work

This section compares AOKell to related projects.

OpenCOM. v1 [8] and v2 [31] is a component model with support for runtime dynamic reconfiguration. OpenCOM supports different kinds of deployment environments (e.g. operating systems, PDAs, embedded devices, network processors) and allows the particularities of those environments to be selectively hidden from or made visible to the OpenCOM programmer. At the application level, OpenCOM components provide interfaces and receptacles (required interfaces). Interceptor components can be associated with interfaces. The architecture of an OpenCOM application is introspectable and can be dynamically modified. Since v2, OpenCOM provides the four following notions: capsule, caplet, loader, and binder. A capsule is a unit of scope that contains and manages the application components. A caplet is a sub-scope within a capsule that contains a subset of the application components. Binders and loaders are first-class entities that provide various ways of binding and loading components. Caplets, loaders and binders are implemented as components, and several implementations may be provided.

Compared to Fractal, capsules and caplets are similar to composite components. Binders and loaders are similar to Fractal controllers. By customizing the implementation of caplets, loaders and binders, programmers have the ability to adapt applications to different deployment environments. The approach is similar in AOKell where controllers are programmed as components. However, we can put forth three differences with OpenCOM. First, AOKell controllers are not restricted to a particular set of functionalities and can implement any kind of services. Second, controllers are components too, but we have gone a step further by introducing the notion of a component architecture at the control level. Finally, the integration of the control dimension and of the functional dimension is achieved with aspects.

Asbaco. [32], like AOKell, is a proposal for extending the membrane of the Fractal components. The authors introduce the term microcomponent to designate a component that implements a control functionality. Like AOKell, Asbaco

microcomponents are associated with the same notions as regular components: they may own client and server interfaces, and bindings can be created between components. However, the API for manipulating bindings between microcomponents is different from the one available for regular components. With AOKell, this API is the same at both levels which leads to a model which is more symmetric. With Asbaco, integrating controllers and regular components is performed with a load-time mixin technique based on the ASM bytecode engineering library [30]. With AOKell, this integration is performed with AspectJ [9]. We believe that the use of AspectJ leads to programs that are easier to write, understand and debug. Although we are currently using the compile-time weaving facility provided by AOKell, we plan to investigate the use of both the compile-time and the load-time features to make the weaving of the control dimension more dynamic.

FuseJ. [6], and JAsCo [23], which is the previous project by the same team, is an architectural description language (ADL) that aims at unifying aspects and components. The FuseJ ADL introduces the notions of a gate and of a connector. A gate, much like an interface in Fractal, is a component communication point. Output and input gates may be defined. Gates are bound to methods provided or required by components. Connectors are responsible for declaratively specifying the architecture of the application. Two kinds of interactions may be specified by connectors: component-based and aspect-oriented. The former case is similar to a binding in Fractal and binds a required gate with a provided one. The latter allows defining an around advice.

Fractal/AOKell and FuseJ differ in the way aspects are used: with AOKell, aspects are only used as a technique for integrating the control and functional dimensions of the component model. The goal of FuseJ is to make aspects first class entities in the component-based programming model. In that sense, FuseJ is similar to another of our project, called FAC [33], which has been build on top of AOKell.

6 Conclusion

This paper presented AOKell, which is a framework for developing component-based applications. AOKell is an implementation of the Fractal Specifications [18] [5]. AOKell is implemented in Java with the AspectJ [9] aspect-oriented language. AOKell has been ported to the .NET platform [10].

Fractal/AOKell provides a component model with two dimensions: the functional and the control dimension. The functional dimension is concerned with the development of application-level functionalities, while the control dimension is concerned with the supervision and the technical services required by the application. While this dichotomy can be found in other component models, e.g. EJB [1] with the notion of a component and of a container, the originality of AOKell is to open the control dimension and to make it programmable. Furthermore, AOKell provides the same concepts for engineering both dimensions. The

notions of a component, of a provided or required interface, and of a binding, are used both to engineer the functional dimension and the control dimension.

AOKell is reflective in the sense that the notion of a component is used both at the functional level and at the control level which can be seen as a kind of meta-level. With AOKell, components are controlled by other, so-called control components. One of the benefits of this approach is to provide a highly dynamic model. By modifying the components assemblies at the control level, programmers can modify the control of their application components. AOKell also enables the precise engineering of the control level. This allows adapting components to execution environments with various needs in term of control, and to foster the development of various forms of control such as the ones needed to program self healing components, self-testing components, or components carrying their proofs or their specifications.

With AOKell, application-level components are controlled by so-called membranes, which are assemblies of control components. Each control component provides a particular control function and may require the services provided by other control components. By componentizing membranes, we foster the reuse, the evolvability and the maintenance of control policies. We then facilitate the development of various control policies, and we obtain a general component model, which can be adapted to application domains with various needs in terms of resources (memory, thread, etc.) and of technical services.

The second originality of AOKell is to use an aspect-oriented approach [19] to integrate the control and the functional dimension of our component model. Each control component is associated with an AspectJ [9] aspect, which is responsible for introducing and supervising the functional component in order to meet the requirement of the control component. In terms of software engineering, this aspect orientation gives a highly expressive solution that facilitates the development and the debugging of the control logic.

As a matter of perspective, we plan to investigate the dynamicity of the relation between a component-based application and its componentized membrane. So far, we have been using the compile-time weaving facility of AspectJ for integration. A load-time weaving mode is also available with AspectJ. Furthermore, other dynamic frameworks are available such as AspectWerkz [21], JAC [22] or JAsCo [23] for runtime weaving. By investigating these solutions, we will be able to provide a fully dynamic model where any modification in the assembly of control components, including features related to interception, will be dynamically applied to the application components without recompilation.

Acknowledgments

This work is partially funded by France Telecom under the external research contract #46131097.

We thank Romain Rouvoy for many discussions about AOKell and for numerous bug reports, Philippe Merle and Renaud Pawlak for their valuable comments about this article.

References

1. Bodoff, S., Armstrong, E., Ball, J., Carson, D.: The J2EE Tutorial. Addison-Wesley (2004) 2nd edition. `java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html`.
2. Siegel, J.: CORBA 3 Fundamentals and Programming. 2nd edn. Wiley (2000)
3. OSGi Alliance: OSGi Technical Whitepaper. (2004) Revision 3.0. `www.osgi.org`.
4. Aldrich, J., Chambers, C., Notkin, D.: ArchJava: Connecting software architecture to implementation. In: Proceedings of the 24th International Conference on Software Engineering (ICSE'02), ACM Press (2002) 187–197
5. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.B.: An open component model and its support in Java. In: Proceedings of the 7th International Symposium on Component-Based Software Engineering (CBSE-7). Volume 3054 of Lecture Notes in Computer Science., Springer (2004) 7–22
6. Suvée, D., Vanderperren, W., Jonckers, V.: FuseJ: An architectural description language for unifying aspects and components. In: Workshop Software-engineering Properties of Languages and Aspect Technologies (SPLAT) at AOSD'05. (2005) `ssel.vub.ac.be/Members/dsuvee/papers/splatsuvee2.pdf`.
7. Dowling, J., Cahill, V.: The K-Component architecture meta-model for self-adaptative software. In: Proceedings of Reflection'01. Volume 2192 of Lecture Notes in Computer Science., Springer-Verlag (2001) 81–88
8. Clarke, M., Blair, G., Coulson, G., Parlavantzas, N.: An efficient component model for the construction of adaptive middleware. In: Proceedings of Middleware'01. (2001)
9. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: Getting started with AspectJ. Communications of the ACM **44**(10) (2001) 59–65
10. Escoffier, C., Donsez, D.: FractNet: An implementation of the Fractal component model for .NET. In: 2ème Journée Francophone sur Développement de Logiciels par Aspects (JFDLPA'05). (2005) `www-adele.imag.fr/fractnet/`.
11. Baude, F., Caromel, D., Morel, M.: From distributed objects to hierarchical grid components. In: Proceedings of the International Symposium on Distributed Objects and Applications (DOA'03). (2003)
12. Fassino, J.P., Stefani, J.B., Lawall, J., Muller, G.: Think: A software framework for component-based operating system kernels. In: Proceedings of the USENIX Annual Technical Conference. (2002) 73–86
13. Rouvoy, R., Merle, P.: Abstraction of transaction demarcation in component-oriented platforms. In: Proceedings of the 4th ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'03). Volume 2672 of Lecture Notes in Computer Science., Springer-Verlag (2003) 305–323
14. ObjectWeb: JORAM: Java open reliable asynchronous messaging. `joram.objectweb.org` (2002)
15. Sun Microsystems: Java Message Service Specification Final Release 1.1. (2002) `java.sun.com/jms`.
16. Alia, M., Chassande-Barrioz, S., Déchamboux, P., Hamon, C., Lefebvre, A.: A middleware framework for the persistence and querying of java objects. In: Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP'04). Volume 3086 of Lecture Notes in Computer Science., Springer-Verlag (2004) 292–316

17. Sun Microsystems: Java Data Objects. (2002) java.sun.com/products/jdo/.
18. Bruneton, E., Coupaye, T., Stefani, J.B.: The Fractal Component Model. ObjectWeb. (2004) Version 2.0.3. fractal.objectweb.org/specification/index.html.
19. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97). Volume 1241 of Lecture Notes in Computer Science., Springer (1997) 220–242
20. Burke, B.: It's the aspects. Java's Developer's Journal (2003) www.sys-con.com/story/?storyid=38104&DE=1.
21. Bonér, J., Dahlstedt, J., Vasseur, A.: AspectWerkz 2: An extensible aspect container. TheServerSide.com (2004) www.theserverside.com/articles/article.tss?l=AspectWerkzP1.
22. Pawlak, R., Seinturier, L., Duchien, L., Florin, G., Legond-Aubry, F., Martelli, L.: JAC: An aspect-based distributed dynamic framework. Software Practice and Experiences (SPE) **34**(12) (2004) 1119–1148
23. Suvée, D., Vanderperren, W., Jonckers, V.: JAsCo: An aspect-oriented approach tailored for component based software development. In: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03), ACM Press (2003) 21–29
24. Sun Microsystems: Enterprise Java Beans. (1997) www.javasoft.com/products/ejb.
25. Fleury, M., Reverbel, F.: The JBoss extensible server. In: Proceedings of the 4th ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'03). Volume 2672 of Lecture Notes in Computer Science., Springer-Verlag (2003) 344–373
26. Seinturier, L., Pessemier, N., Coupaye, T.: AOKell: An aspect-oriented implementation of the Fractal specifications. Objectweb Fractal Workshop, Grenoble, France (2005)
27. Szyperski, C.: Component Software - Beyond Object-Oriented Programming. 2nd edn. Addison-Wesley (2002)
28. Leclercq, M., Quema, V., Stefani, J.B.: DREAM: a component framework for the construction of resource-aware, configurable middleware. IEEE Distributed Systems Online **6**(9) (2005)
29. Gil, T., Evain, J.B.: AspectDNG. DotNetGuru. (2005) www.dotnetguru.biz/aspectdng/.
30. Bruneton, E., Lenglet, R., Coupaye, T.: ASM: A code manipulation tool to implement adaptable systems. In: Journées Composants 2002 (JC'02). (2002) asm.objectweb.org/current/asm-eng.pdf.
31. Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., Uyema, J.: A component model for building systems software. In: Proceedings of the IASTED Software Engineering and Applications (SEA'04). (2004)
32. Mencl, V., Bures, T.: Microcomponent-based component controllers: A foundation for component aspects. In: Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05). (2005)
33. Pessemier, N., Seinturier, L., Duchien, L., Coupaye, T.: A model for developing component-based and aspect-oriented systems. In: Proceedings of the 5th International Symposium on Software Composition (SC'06). Lecture Notes in Computer Science, Springer (2006)